OSProj2 UNIX Shell Programming & Linux Kernel Module for Task Information

by 潘禧辰，518021910497

Menu

OSProj2 UNIX Shell Programming & Linux Kernel Module for Task Information
- Menu
- Abstract
- Environment
- Quick Start
  - 编译
    - gcc编译
    - Makefile的编写
  - 测试代码
    - Simple Unix Shell
    - Linux Kernel Module
- Implementation & Result
  - Simple Unix Shell
    - 初始化
    - 循环读入
    - 判断
    - 执行
    - 完整代码
    - 结果
  - Linux Kernel Module
    - 结果
- Difficulties
- Reference

Abstract

- 编写了一个简单的Unix shell，完成了子进程执行指令、执行历史指令、输入输出重定向、允许父子进程通过pipe交流的功能
- 编写了一个Linux Kernel Module根据pid输出对应进程的指令、进程号和状态

Environment

- Ubuntu 18.04
- Linux 5.3.0-42-generic
- VMware Workstation Rro 15.5.0 build-14665864

Quick Start

编译

gcc编译

simple Unix shell是用户态代码，直接使用如下gcc命令进行编译。

```
gcc proj2-1.c -o proj2-1
```

Makefile的编写

Linux Kernel Module是内核态代码，需要编写Makefile，编写方法与Project1类似。

```
obj-m := pid.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

测试代码

Simple Unix Shell

使用以下代码对四个功能进行测试

```
gcc proj2-1.c -o proj2-1        # 进行编译
./proj2-1                       # 打开编译后文件
osh>ls                          # 使用ls指令测试能否完成指令执行
osh>!!                          # 测试能否执行历史指令
```

```
./proj2-1                       # 打开编译后文件
osh>!!                          # 测试历史指令为空时，能否正确输出NO commands in
history.
osh>ls > out.txt                # 测试能否进行输出重定向
osh>sort < in.txt               # 测试能否进行输入重定向
osh>ls -l | less                # 测试能否进行父子进程communicate
```

其中`in.txt`内容如下

```
5
4
3
2
1
```

Linux Kernel Module

```
make                            # 进行编译
sudo insmod pid.ko              # 加载pid内核模块
echo 1395 > /proc/pid           # 写入进程号
cat /proc/pid                   # 显示对应进程的指令、进程号和状态
sudo rmmod pid                  # 删除pid内核模块
```

Implementation & Result

Simple Unix Shell

整体的思路是初始化后先依次对指令进行读入，依次判断是否出现`exit`、`!!`、`&`、`|`、`>`、`<`这些在Simple Unix Shell中特殊约定的，`execvp`函数无法直接执行的字符。对这些字符进行识别后将相应的功能flag做修改方便后边对应执行，并且删去这些字符。

初始化

初始化先短暂sleep使得父进程不wait子进程时仍然可以正常输出。之后清空`args`字符串指针，并将标记指令长度的变量`pos`，标记pipe通信时另一指令长度的变量`pipe_pos`，标记是否需要等待的变量`wait_flag`，标记是否需要进行重定向的变量`red_flag`，标记是否需要进行pipe通信的变量`pipe_flag`都进行初始化

```
        usleep(10000);
        printf("osh>");
        fflush(stdout);
        for (int i = 0; i < MAX_LINE/2 + 1; i++) //初始化清空args
            args[i] = NULL;
        //初始化变量
        pos = 0;
        pipe_pos = 0;
        wait_flag = 1;
        red_flag = 0;
        pipe_flag = 0;
```

循环读入

使用以下代码进行指令读入，每次读入一个argument，同时读入空格，最终遇到换行符时结束读入

```
        //循环读入指令
        while (scanf("%s", str[pos]))
        {
            args[pos] = str[pos];
            scanf("%c", &ch);
            pos++;
            if(ch == '\n')
                break;
        }
```

判断

这部分分别对`exit`、`!!`、`&`、`|`四种字符进行了识别和清除。

在判断历史指令时发现如果不是`!!`则在`history_args`中储存指令，如果是`!!`而此时历史指令为空，那么就输出`NO commands in history.`同时进入下个循环，如果不为空，那么将当前指令修改为历史指令。

在判断重定向时，根据指令修改`red_flag`并且将文件名存入字符串指针`filename`

在判断pipe通信时，如果有pipe通信，那么将另一进程的指令存入`pipe_args`

```
        // 判断结束
        if (strcmp(args[0], "exit") == 0)
        {
            should_run = 0;
```

```
            continue;
        }
        // 判断历史指令
        if (strcmp(args[0], "!!") != 0)
        {
            for (int i = 0; i < pos; i++)
            {
                strcpy(temp, args[i]);
                history_args[i] = temp;
            }
            history_pos = pos;
        }
        else{
            if (history_args[0] == NULL) {
                printf("NO commands in history.\n");
                continue;
            }
            else {
                for (int i = 0; i < history_pos; i++) {
                    strcpy(temp, history_args[i]);
                    args[i] = temp;
                }
                pos = history_pos;
            }
        }
        //判断wait
        if (strcmp(args[pos - 1], "&") == 0)
        {
            wait_flag = 0;
            args[pos - 1] = NULL;
            pos--;
        }
        //判断重定向
        if (pos>=2 && strcmp(args[pos-2],">")==0)
        {
            red_flag = 1;
            strcpy(temp, args[pos-1]);
            filename = temp;
            args[pos-1] = NULL;
            pos--;
            args[pos-1] = NULL;
            pos--;
        }
        else if (pos>=2 && strcmp(args[pos-2],"<")==0)
        {
            red_flag = 2;
            strcpy(temp, args[pos-1]);
            filename = temp;
            args[pos-1] = NULL;
            pos--;
            args[pos-1] = NULL;
            pos--;
        }
        //判断 pipe
        for (int i = 0; i < pos; i++)
        {
            if (strcmp(args[i], "|") == 0)
            {
```

```
                pipe_flag = 1;
                args[i] = NULL;

                for (int j = i + 1; j < pos; j++)
                {
                    strcpy(temp, args[j]);
                    pipe_args[pipe_pos] = temp;
                    args[j] = NULL;
                    pipe_pos++;
                }
                pos -= pipe_pos;
                break;
            }
        }
```

执行

执行部分先进行`fork`，在子进程进行指令执行，父进程根据之前的`wait_flag`选择等待调用`wait(NULL)`或者不等待直接结束调用`signal(SIGCHLD, SIG_IGN)`。

子进程中先根据之前的`red_flag`建立输入输出重定向，如果存在重定向使用`open`打开文件，使用`dup2`将文件复用至`stdout`或`stdin`达到重定向的目的。

再根据之前的`pipe_flag`建立通信：父进程即`|`前的进程，执行`args`的代码，并且将输出重定向至`pipe.txt`；子进程即`|`后的进程，先延时0.5s等待父进程输出，再执行`pipe_args`的代码，并且将输入重定向至`pipe.txt`，完成pipe通信。

```
        //执行
        pid = fork();
        if (pid==0)//子进程
        {
            //建立重定向
            if (red_flag == 1)
            {
                fd = open(filename,O_CREAT|O_RDWR,S_IRWXU);
                dup2(fd, STDOUT_FILENO);
            }
            else if (red_flag == 2)
            {
                fd = open(filename,O_CREAT|O_RDONLY,S_IRUSR);
                dup2(fd, STDIN_FILENO);
            }
            //建立通信
            if(pipe_flag)
            {
                pid_t pipe_pid;
                pipe_pid = fork();
                if(pipe_pid == 0)// 子进程执行pipe_arg
                {
                    usleep(500000);
                    fd = open("pipe.txt",O_CREAT|O_RDONLY,S_IRUSR);
                    dup2(fd, STDIN_FILENO);
                    execvp(pipe_args[0], pipe_args);
                    exit(0);
                }
                else// 父进程
                {
```

```
                    fd = open("pipe.txt",O_CREAT|O_RDWR,S_IRWXU);
                    dup2(fd, STDOUT_FILENO);
                    execvp(args[0], args);
                    wait(NULL);
                }
            }
            else
                execvp(args[0], args);
            exit(0);
        }
        else//父进程
        {
            if(wait_flag)
                wait(NULL);
            else
                signal(SIGCHLD, SIG_IGN);
        }
```

完整代码

将上述几个部分进行简单组合放入`should_run`循环就可以了。

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdlib.h>
#include <signal.h>

#define MAX_LINE 80 /* The maximum length command */

char *args[MAX_LINE/2 + 1]; /* command line arguments */
char *history_args[MAX_LINE/2 + 1];
char *pipe_args[MAX_LINE/2 + 1];
char *filename;
char temp[20];
int should_run = 1; /* flag to determine when to exit program */
int pos;
int history_pos;
int pipe_pos;
char ch;
char str[MAX_LINE][20];
int wait_flag;
int red_flag; //0为不需要重定向，1为输出重定向，2为输入重定向
int pipe_flag;
int fd;
pid_t pid;
pid_t pipe_pid;

int main(void)
{
    while (should_run) {
        usleep(10000);
        printf("osh>");
        fflush(stdout);
```

```
for (int i = 0; i < MAX_LINE/2 + 1; i++) //初始化清空args
    args[i] = NULL;
//初始化变量
pos = 0;
pipe_pos = 0;
wait_flag = 1;
red_flag = 0;
pipe_flag = 0;
//循环读入指令
while (scanf("%s", str[pos]))
{
    args[pos] = str[pos];
    scanf("%c", &ch);
    pos++;
    if(ch == '\n')
        break;
}
// 判断结束
if (strcmp(args[0], "exit") == 0)
{
    should_run = 0;
    continue;
}
// 判断历史指令
if (strcmp(args[0], "!!") != 0)
{
    for (int i = 0; i < pos; i++)
    {
        strcpy(temp, args[i]);
        history_args[i] = temp;
    }
    history_pos = pos;
}
else{
    if (history_args[0] == NULL) {
        printf("NO commands in history.\n");
        continue;
    }
    else {
        for (int i = 0; i < history_pos; i++) {
            strcpy(temp, history_args[i]);
            args[i] = temp;
        }
        pos = history_pos;
    }
}
//判断wait
if (strcmp(args[pos - 1], "&") == 0)
{
    wait_flag = 0;
    args[pos - 1] = NULL;
    pos--;
}
//判断重定向
if (pos>=2 && strcmp(args[pos-2],">")==0)
{
    red_flag = 1;
```

```
            strcpy(temp, args[pos-1]);
            filename = temp;
            args[pos-1] = NULL;
            pos--;
            args[pos-1] = NULL;
            pos--;
        }
        else if (pos>=2 && strcmp(args[pos-2],"<")==0)
        {
            red_flag = 2;
            strcpy(temp, args[pos-1]);
            filename = temp;
            args[pos-1] = NULL;
            pos--;
            args[pos-1] = NULL;
            pos--;
        }
        //判断 pipe
        for (int i = 0; i < pos; i++)
        {
            if (strcmp(args[i], "|") == 0)
            {
                pipe_flag = 1;
                args[i] = NULL;

                for (int j = i + 1; j < pos; j++)
                {
                    strcpy(temp, args[j]);
                    pipe_args[pipe_pos] = temp;
                    args[j] = NULL;
                    pipe_pos++;
                }
                pos -= pipe_pos;
                break;
            }
        }
        //执行
        pid = fork();
        if (pid==0)//子进程
        {
            //建立重定向
            if (red_flag == 1)
            {
                fd = open(filename,O_CREAT|O_RDWR,S_IRWXU);
                dup2(fd, STDOUT_FILENO);
            }
            else if (red_flag == 2)
            {
                fd = open(filename,O_CREAT|O_RDONLY,S_IRUSR);
                dup2(fd, STDIN_FILENO);
            }
            //建立通信
            if(pipe_flag)
            {
                pipe_pid = fork();
                if(pipe_pid == 0)// 子进程执行pipe_arg
                {
                    usleep(500000);
```

```
                    fd = open("pipe.txt",O_CREAT|O_RDONLY,S_IRUSR);
                    dup2(fd, STDIN_FILENO);
                    execvp(pipe_args[0], pipe_args);
                    exit(0);
                }
                else// 父进程
                {
                    fd = open("pipe.txt",O_CREAT|O_RDWR,S_IRWXU);
                    dup2(fd, STDOUT_FILENO);
                    execvp(args[0], args);
                    wait(NULL);
                }
            }
            else
                execvp(args[0], args);
            exit(0);
        }
        else//父进程
        {
            if(wait_flag)
                wait(NULL);
            else
                signal(SIGCHLD, SIG_IGN);
        }
/**
* After reading user input, the steps are:
* (1) fork a child process using fork()
* (2) the child process will invoke execvp()
* (3) parent will invoke wait() unless command included &
*/
    }
    return 0;
}
```

结果

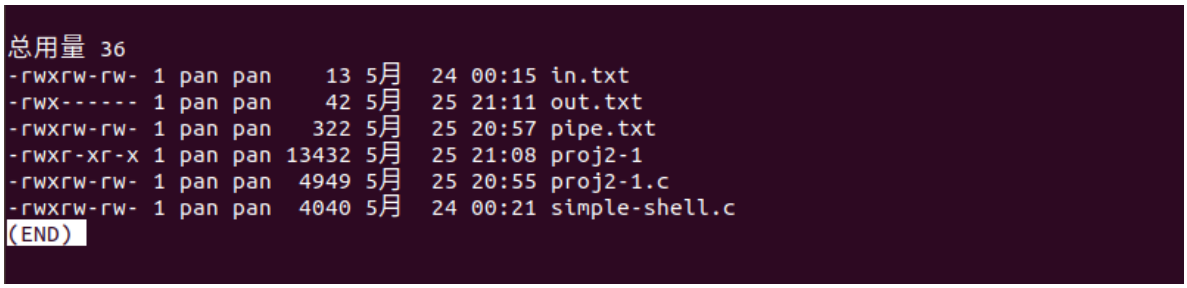
- 测试指令执行和历史指令执行

```
pan@pan-virtual-machine:~/桌面/osproj/2/1$ gcc proj2-1.c -o proj2-1
pan@pan-virtual-machine:~/桌面/osproj/2/1$ ./proj2-1
osh>ls
in.txt  pipe.txt  proj2-1  proj2-1.c
osh>!!
in.txt  pipe.txt  proj2-1  proj2-1.c
osh>
```

- 测试历史指令为空时，能否正确输出NO commands in history.，测试能否进行输入输出重定向

```
pan@pan-virtual-machine:~/桌面/osproj/2/1$ ./proj2-1
osh>!!
NO commands in history.
osh>ls > out.txt
osh>sort < in.txt
1
2
3
4
5
osh>
```

- 测试能否进行父子进程communicate



Linux Kernel Module

根据课本提供的代码添加了变量`cupid`用于记录进程号，在`proc_write`中使用`kstrtol(k_mem, 10, &cupid);`指令完成进程号的获取

在`proc_read`中使用`Task_struct = pid_task(find_vpid(cupid), PIDTYPE_PID);`获取进程的指令、进程号和状态。之后再输出到`buffer`中

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>
#include <linux/sched.h>
#include <linux/slab.h>
#define BUFFER_SIZE 128
#define PROC_NAME "pid"

static long cupid;

static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t
count, loff_t *pos);
static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t count,
loff_t *pos);

static struct file_operations proc_ops = {
        .owner = THIS_MODULE,
        .read = proc_read,
        .write = proc_write,
};
/* This function is called when the module is loaded. */
static int proc_init(void)
{
    /* creates the /proc/hello entry */
    proc_create(PROC_NAME, 0666, NULL, &proc_ops);
    return 0;
}
```

```c
/* This function is called when the module is removed. */
static void proc_exit(void)
{
    /* removes the /proc/hello entry */
    remove_proc_entry(PROC_NAME, NULL);
}

// write
static ssize_t proc_write(struct file *file, const char __user *usr_buf, size_t
count, loff_t *pos)
{
    char *k_mem;
    /* allocate kernel memory */
    k_mem = kmalloc(count, GFP_KERNEL);
    /* copies user space usr buf to kernel memory */
    copy_from_user(k_mem, usr_buf, count);
    /* return kernel memory */
    kstrtol(k_mem, 10, &cupid);
    kfree(k_mem);
    return count;
}

// read
static ssize_t proc_read(struct file *file, char __user *usr_buf, size_t count,
loff_t *pos)
{
    int rv = 0;
    char buffer[BUFFER_SIZE];
    static int completed = 0;
    struct task_struct *Task_struct;

    if (completed) {
        completed = 0;
        return 0;
    }
    Task_struct = pid_task(find_vpid(cupid), PIDTYPE_PID);
    if (Task_struct == NULL)
        rv = sprintf(buffer, "%d\n", 0);
    else
        rv = sprintf(buffer, "command = [%s] pid = [%d] state = [%ld]\n",
Task_struct -> comm,  Task_struct -> pid, Task_struct -> state);
    /* copies kernel space buffer to user space usr buf */
    copy_to_user(usr_buf, buffer, rv);
    completed = 1;
    return rv;
}

module_init(proc_init);
module_exit(proc_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Proj2");
MODULE_AUTHOR("Xichen Pan");
```

结果

```
pan@pan-virtual-machine:~/桌面/osproj/2/2$ make
make -C /lib/modules/5.3.0-42-generic/build M=/home/pan/桌面/osproj/2/2 modules
make[1]: 进入目录“/usr/src/linux-headers-5.3.0-42-generic”
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: 离开目录“/usr/src/linux-headers-5.3.0-42-generic”
pan@pan-virtual-machine:~/桌面/osproj/2/2$ sudo insmod pid.ko
[sudo] pan 的密码：
pan@pan-virtual-machine:~/桌面/osproj/2/2$ echo 1 > /proc/pid
pan@pan-virtual-machine:~/桌面/osproj/2/2$ cat /proc/pid
command = [systemd] pid = [1] state = [1]
pan@pan-virtual-machine:~/桌面/osproj/2/2$ sudo rmmod pid
pan@pan-virtual-machine:~/桌面/osproj/2/2$
```

Difficulties

- 因为Simple Unix Shell是用户态代码，最初没有使用gcc编译，而仍然使用Makefile，出现了无法编译成功的问题。最初认为是系统源码的问题，对系统进行了很多次的修补，最终才找到原因。
- Linux Kernel Module部分因为直接在内核态运行，出现了bug之后往往在rmmod时发生卡死，只能重新利用快照回滚和重启系统，debug阶段花费了非常多的时间。

Reference

- Operating System Concept 10^{th} edition